Agenda: Working with Classes

- Writing and Using Classes
- Constructor method
- Inheritance of classes
- Type casting
- Type Assertion
- Static Properties
- Abstract class

Working with Classes

Starting with ECMAScript 2015, also known as ECMAScript 6, JavaScript programmers were able to build their applications using this object-oriented class-based approach.

TypeScript allow developers to use these techniques now, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript.

Writing Class:

1. Person class with following members:
    a. Fields: firstName, lastName, dateOfBirth,
    b. constructor,
    c. Methods: greet, selfIntroduction.

```
class Person {
  firstName: string
  lastName: string
  dateOfBirth: Date
  constructor(fn: string, ln: string, dateOfBirth: Date) {
    this.firstName = fn;
    this.lastName = ln;
    this.dateOfBirth = dateOfBirth;
  }
  greet(friendName: string): string {
    return "Hello " + friendName + ", how are you doing?";
  }
  selfIntrodution(): string {
    let age = (new Date().getFullYear() - this.dateOfBirth.getFullYear())
    return "Hello, I am " + this.firstName + " " + this.lastName + " and I am  " + age + "yrs old";
  }
}
```

Note: "this" is compulsory for every member when used inside a method.

Using Class:

```
let person: Person = new Person("Sandeep", "Soni");
alert(person.greet("Rahul"));
```

constructor and properties

```
class Person {
  constructor(private firstName: string, private lastName: string) {
  }
}
```

Is same as

```
class Person {
  private firstName: string;
  private lastName: string;
  constructor(fn: string, ln: string) {
    this.firstName = fn;
    this.lastName = ln;
  }
}
```

Inheritance: The following example demonstrates:

2. Extending class
    a. Class Employee in extended/inherited from Person
        i. Fields: department, salary
        ii. Constructor MUST call Parent class constructor.

```
class Employee extends Person {
  departmentName: string
  salary: number
  constructor(fn: string, ln: string, dateOfBirth: Date, deptName: string, sal: number) {
    super(fn, ln, dateOfBirth);
    this.departmentName = deptName;
    this.salary = sal;
  }
}
```

Note: Every Child class constructor must call Parent class constructor using *super*.

3. Method overriding and super to call parent class method.
    a. **Override self-Introduction**

```
selfIntrodution(): string {
    var s: string = super.selfIntrodution()
    return s + " working in " + this._departmentName + " department";
}
```

4. **Public, private and protected, read-only modifiers**

accessible on	public	protected	private
class	yes	yes	yes
class children	yes	yes	no
class instances	yes	no	no

    a. By default all members are public
    b. Person class:
        i. Change firstName and lastName to private.
        ii. Change dateOfBirth as protected
        iii. Change dateOfBirth as readonly. Readonly Members can be initialized in constructor only.

5. **get and set Accessors**
    a. In Person class
        i. Add only getter for age.
        ii. Add get accessor fullName, returning firstName + " " + lastName
    b. Employee Class
        i. Add incrementSalary method
        ii. Change salary to readonly.
        iii. Validate departmentName not to allow ""

```
class Person {
  protected firstName: string
  protected lastName: string
  private readonly dateOfBirth: Date

  public get fullName(): string {
    return this.firstName + this.lastName;
  }
  public get age(): number {
    return (new Date().getFullYear() - this.dateOfBirth.getFullYear());
  }
  constructor(fn: string, ln: string, dateOfBirth: Date) {
    this.firstName = fn;
    this.lastName = ln;
    this.dateOfBirth = dateOfBirth;
  }
  greet(friendName: string): string {
    return "Hello " + friendName + ", how are you doing?";
  }
  selfIntrodution(): string {
    let age = (new Date().getFullYear() - this.dateOfBirth.getFullYear())
    return "Hello, I am " + this.firstName + " " + this.lastName + " and I am  " + age + "yrs old";
  }
}

class Employee extends Person {
  private _departmentName: string
  private _salary: number
  constructor(fn: string, ln: string, dateOfBirth: Date, deptName: string, sal: number) {
    super(fn, ln, dateOfBirth);
    this._departmentName = deptName;
    this._salary = sal;
  }
  //Method overriding
  selfIntrodution(): string {
    var s: string = super.selfIntrodution()
    return s + " working in " + this._departmentName + " department";
  }
  public incrementSalary(amount: number) {
    this._salary += amount;
  }
  public get salary(): number {
    return this._salary;
  }
  public get departmentName(): string {
        return this._departmentName
    }
    public set departmentName(newValue: string) {
        if (newValue == "")
            throw "Department cannot be null";
        this._departmentName = newValue;
    }
}
```

6. Type casting

Use **< >** or as keyword for casting. Since Typescript 1.6, the default is as because **< >** is ambiguous in **.jsx** files.

Using **Instanceof** you can find out the type of any object, similarly using **typeof** you can find out the type of a variable.

Ex:

```
var obj1: string;
var obj2: any;
obj1 = <string>obj2;    // using <>
obj1 = obj2 as string;   // using as keyword
```

```
let person: Person
person = emp;
emp = <Employee>person;
```

7. Type Assertion

Overriding / customizing the types of typescript in any way you want to do is known as Type Assertion mechanism.

Here is the reason why we are choosing for Type Assertion.

Javascript / typescript does not allow to create dynamic properties to any object i.e. we cannot expand the properties in any object dynamically

```
var user = {};
user.firstName = "Hello"   //Error: property firstName does not exist on type {}
```

We can fix this simply by using type assertion **as IUser**

```
interface IUser {
userId: number;
userName: string;
}
```

```
var user = {} as IUser;
user.userId = 123;
user.userName = 'hello';
```

Here the compiler will provides autocomplete for the object properties.

```
var user = <IUser>{
// the compiler will provide autocomplete for properties of IUser
};
```

8. Static Properties:

a. Must be accessed using class name.

b. Add MaxSalary as static member

c. Change incrementSalary to validate new salary.

Edit class Employee

```
public static MaxSalary: number;
public incrementSalary(amount: number) {
    if (this._salary + amount > Employee.MaxSalary)
      throw "Salary cannot be greater than " + Employee.MaxSalary;
    this._salary += amount;
  }
```

9. Abstract class:

- Abstract classes are base classes from which other classes may be derived.
- They may not be instantiated directly.
- An abstract class may contain implementation details for its members.

```
abstract class Figure
{
  protected Dimension: number
  public abstract Area(): number
}

class Circle extends Figure
{
  constructor(radius: number)  {
    super();
    this.Dimension = radius;
  }
  public Area(): number {
    return Math.PI * this.Dimension * this.Dimension;
  }
}

class Square extends Figure {
  constructor(side: number)  {
    super();
    this.Dimension = side;
  }
  public Area(): number {
    return this.Dimension * this.Dimension;
  }
}

var fig: Figure = new Circle(10); //new Square(10);
alert(fig.Area());
```